LOAN DOCUMENT

PHOTOGRAPH THIS SHEET

DTIC ACCESSION NUMBER

LEVEL

D

INVENTORY

RIA-77-U990

DOCUMENT IDENTIFICATION

JUN 72

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR
NTIS GRA&I
DTIC TRAC
UNANNOUNCED
JUSTIFICATION

BY
DISTRIBUTION/
AVAILABILITY CODES

DISTRIBUTION	AVAILABILITY AND/OR SPECIAL	
A-1		

DISTRIBUTION STAMP

DATE ACCESSIONED

DATE RETURNED

19990517 104

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NUMBER

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-FDAC

H A N D L E W I T H C A R E

DTIC FORM 70A JUN 90

DOCUMENT PROCESSING SHEET

PREVIOUS EDITIONS MAY BE USED UNTIL STOCK IS EXHAUSTED.

LOAN DOCUMENT

N73-11189

DEBUGGING COMPUTER PROGRAMS: A SURVEY WITH SPECIAL EMPHASIS ON ALGOL

R. S. Scowen

National Physical Laboratory
Teddington, Middlesex, England

June 1972

See note inside cover

NPL Report NAC 21
June 1972

(NPL-NAC-21) DEBUGGING COMPUTER PROGRAMS.
A SURVEY WITH SPECIAL EMPHASIS ON ALGOL
R.S. Scowen (National Physical Lab.) Jun.
1972 39 p N73-11189

Unclas
H2/08 91360

National Physical Laboratory

Division of Numerical Analysis and Computing

DEBUGGING COMPUTER PROGRAMS
A SURVEY WITH SPECIAL EMPHASIS
ON ALGOL

by R.S. Scowen

NOTICE

DEBUGGING COMPUTER PROGRAMS
A SURVEY WITH SPECIAL EMPHASIS ON ALGOL

R. S. Scowen
National Physical Laboratory
Teddington, Middlesex, England

Abstract

This report considers the problems of debugging computer programs and some of the tools which can simplify the task. The main sections describe:- (1) ways in which compilers can aid the debugging of programs, (2) ALGOL programs which can be used to determine the kinds of errors detected by an ALGOL compiler, (3) results of running the test programs on sixteen different compilers, (4) results obtained from a survey of the errors made by programmers at N.P.L.

CONTENTS

INTRODUCTION

Debugging is the process of locating and correcting the errors in a computer program; the efficiency with which it can be carried out depends critically on the compilers and software available. This report describes:- (1) some properties which compilers should possess to simplify debugging; (2) a number of small ALGOL 60 test programs designed to discover how helpfully an ALGOL compiler treats incorrect programs; (3) a summary of the results which were obtained when the test programs were run on sixteen different compilers; (4) the results of a survey of failures in ALGOL 60 programs recorded by suitably modified compilers used at NPL.

Debugging would be unnecessary if programs could be proved to be correct. Much work is going on in this field (see London 1970, and Adams et al, 1972) but the techniques are not yet widely applied, and debugging is likely to be necessary for some time to come. The topic is not considered further in the present report, neither are the special problems of debugging real-time systems; see a paper by van Horn (1968) for some pertinent suggestions.

Debugging tools can be classified as active or passive. An active tool is one which enables the programmer to specify what he wants after he has realized there is an error; examples are given below in the paragraph 'Extra debugging facilities' on page 4. A passive tool works automatically without any effort from the programmer, e.g. failure messages, store post-mortems, etc. This report is mainly concerned with passive tools since these are generally more useful. The basic handicap of active debugging aids is that they rely on foreknowledge of where the errors will occur.

USEFUL PRINCIPLES

Detecting errors

A compiler should not translate and run illegal programs. It not only makes debugging harder but also adds to the difficulty of ensuring that a program is machine independent.

Errors should be found during translation rather than at runtime; less computer time is then wasted and the failure message is more likely to be helpful because the position of the error can be specified more precisely. Errors which are not found by the compiler are particularly wasteful of the programmer's time; the only evidence is often the whole program and a mass of more or less incorrect results. With programs that lose control (e.g. by overwriting the program or compiler) the only evidence is an octal or hexadecimal core-dump (see Anon 1969).

Of course, not all programming errors can be found by a compiler. For example, if the programmer writes the constant '997' instead of '977', or uses faulty logic (e.g. see Forsythe 1970), then the program is legal but performing the wrong task. Occasionally the compiler can help detect these errors by printing warning messages when it finds odd features in the program.

Error messages

Failure messages should be intelligible; if the programmer cannot understand them, then effectively all he is told is 'invalid program'. Ideally all messages should be in a language understood by the programmer. It is a poor compiler if the programmer needs to know the assembly or machine language in order to be able to debug his programs. Error messages should also be reliable and not tell the programmer that he has made one sort of error when, in fact, the mistake is something quite different.

When the compiler finds a syntax error, it should report the position and cause of the error precisely and clearly. Note that the position of the error may not be where it is discovered. For example, although the KDF9 Whetstone ALGOL compiler does not detect 'variable used but not declared' until the end of the translation, it nevertheless tells the programmer where the variable was used.

One concise way of specifying the position is to give a line number, but this will be insufficient unless the programmer can easily identify the line concerned. There are other methods of specifying the position of an error clearly, e.g. the compiler can print the symbols which occur just before and after the error, or give the number of lines since the start of the latest procedure declaration or label. The clearest way is to print a listing of the program with the error messages interspersed in the appropriate places. However this can be expensive, it is also often impractical in an online situation.

When an error is discovered during execution the minimum amount of

information which should be given is:-

(1) The cause of the error.
(2) The position in the program text where the error was detected. The position in the object code is less useful.

Not all compilers help even this much. In any case extra information is nearly always useful, e.g.

(3) The route of the program just before it failed (a 'retroactive trace' in KDF9 terminology).
(4) The value of some or all the variables at the time the program failed.

This information can be printed only if runtime errors are discovered before the whole store has been corrupted.

Efficiency

Some compilers contain optional facilities which test the program more thoroughly. If these aids are slow and expensive, the programmer may be unable to afford to use them. The KDF9 ALGOL system is faulty in this respect. It consists of two compatible compilers: one (Whetstone) is designed for program testing, and the other (Kidsgrove) for executing correct programs. Unfortunately the Whetstone compiler executes programs so slowly that some programs have to be debugged using the Kidsgrove compiler. Also the system will be inefficient unless as many errors as possible are found during each translation. It is impossible to find every syntax error in every program but a good compiler should find most of the errors most of the time. Note that successful recovery from an error is more difficult in languages with a nested recursive structure like ALGOL than in one-statement-at-a-time languages like FORTRAN or BASIC.

Whether a program should be executed further after a runtime error is debatable. Additional errors might be discovered, but the program is going to run for a longer time and cost more; the results are bound to be wrong and it may be more difficult to trace the first error because some evidence will have been destroyed. Perhaps the compiler should continue after some runtime errors (e.g. when a value does not fit a specified output format), but not after others.

A useful option for load-and-go compilers (*1) would allow a program to be executed more than once with different sets of data even if it fails at runtime.

---

(1) A load-and-go compiler compiles a program and immediately executes it

Extra debugging options

Debugging can often be simplified if extra facilities are available to the programmer.

1. Tracing

Tracing is the process of printing a record of the steps executed by a program while it is running. Tracing is rarely needed if other debugging facilities are good, but occasionally it is extremely valuable; various levels are useful, e.g:-

(1) Every procedure call or label

(2) Every jump

(3) Every assignment statement

(4) Every operation

It is also helpful if the values of some or all of the variables can be printed when tracing is switched on.

Tracing is not so useful if it can be switched on and off only during the translation and thus must be performed every time the specified parts of the program are executed. Difficulties will arise when an error occurs on the last time round a loop. Printing the trace each time round the loop would be very slow and expensive and so it is essential to be able to trace only the last few relevant circuits.

A variant of tracing is the option of printing the value of a particular variable every time there is an assignment to it.

2. Documentation aids

Documentation programs [e.g. SOAP (Scowen et al, 1971), NEATER/2 (Conrow 1970) and flowcharters] list a program in a consistent way which clarifies its structure and action. Flowcharters have the disadvantage that they present all parts of the program with the same degree of emphasis, rather like a map that shows footpaths and motorways in the same way. SOAP is generally superior for documenting ALGOL programs; by indenting some lines more than others it exposes the extent of any statement, declaration or comment, and the alternatives in a conditional statement. SOAP is also faster and cheaper than flowcharters.

3. A flow-trace

A flow-trace is a listing or table which specifies how many times each part of a program has been executed. It too is a useful tool for gaining insight into what a program does.

A related facility gives the amount of time spent in each part of the program.

4. A concordance

When a programmer has to modify a large program, he will often find he is unable to understand the use of a particular variable; in these circumstances he needs a concordance of the program, i.e. a listing of the declaration, uses and assignments of every variable in the program.

5. A general macro processor

A general macro language and processor (e.g. ML/I, see Brown 1966,1967) is useful when altering large progams from one language or dialect to another. A general macro-processor will make many of the alterations consistently and avoid introducing random trivial mistakes.

The operating system

The characteristics of a compiler should depend on the sort of operating system in which it is embedded. For a multi-access operating system with remote job entry facilities, the compiler should be small and fast. Only a limited amount of output is desirable if the printing speed of the terminal is slow; certainly there is no time for a program listing or reference tables. Output on a display or line-printer should still be brief; too much is merely confusing. Some systems continue to print information which was needed by the implementors when the system was being developed. Also it is less important to discover all the syntax errors during one translation because little will be lost if an extra compilation is necessary.

On the other hand, for a batch operating system with a turn-round measured in hours, the programmer will feel frustrated if the compiler discovers only one error in each run. It may be appropriate to execute a program up to the first syntax error even if it fails to translate (as in 1900 ALGOL).

Debugging and programming language design

Two of the most important objectives in designing new programming languages should be to make it easier to write programs and less easy to make mistakes. These objectives are often not given their due importance but they can be achieved. A concise natural notation using simple consistent rules helps the programmer avoid errors; and redundancy in the language ensures that as many errors as possible are found during translation. Default rules are dangerous because they can be applied unwittingly; it is safest and simplest if programmers follow the advice given to Alice: they say what they mean as well as mean what they say.

The modern versions of ALGOL are superior to ALGOL 60 which itself has advantages over FORTRAN. Some common programming errors, which would be more easily cured or avoided if FORTRAN had been defined differently, are described by Elspas et al (1971) and by Evershed et al (1971). DITRAN (see Moulton et al 1967) is an implementation of FORTRAN which does a full check on the legality of a program. The authors wanted to use the compiler for teaching students and

they were more interested in fast translation and good diagnostics than in runtime speed. The paper describes the contortions which are essential if a complete check is to be performed on a FORTRAN program but it does not state what effect the checks have on runtime efficiency.

Debugging and compiler design

The best and most efficient compilers for debugging purposes are probably those which are load-and-go and produce machine code (e.g. ALGOL W, Babel, WATFOR). Load-and-go compilers are convenient to use because there is only a single job to be submitted to the operating system. Compilers are faster when they produce machine code because they avoid the overheads of assemblers and linkage editors.

It is a common belief that an interpretive system is necessary for good runtime diagnostics, but this is not true. It is not difficult to produce and store tables during compilation which can be used after a runtime failure to interpret and output sensibly the contents of the store. Other debugging facilities can be provided by compiling programs in slightly different ways.

THE ALGOL TEST PROGRAMS

1. Programs which should fail to translate

1.1 Illegal syntax

```
begin
        real ;
end
```

1.2 Variable used but not declared

```
begin
        x := 2.3
end
```

1.3 Variable declared twice

```
begin
        real x;
        integer x;
end
```

1.4 Invalid operator

```
begin
        real x, y;
        x := y > x
end
```

1.5 Wrong number of subscripts

```
begin
        real x;
        array a[1 : 10];
        x := a[1, 3]
end
```

1.6 Actual parameter of wrong type

```
begin
        procedure p(x);
                real x;
                ;
        boolean z;
        p(z)
end
```

1.7 Illegal use of constant as parameter

```
begin
        procedure p(x);
                real x;
                x := 3.14;
        p(2.71)
end
```

1.8 Wrong number of parameters

```
begin
        procedure p(x);
                real x;
                ;
        p(2, 4)
end
```

1.9 Wrong number of subscripts in a formal array

```
begin
        procedure p(a);
                array a;
                a[1] := 0;
        array a[0 : 3, 0 : 3];
        p(a)
end
```

1.10 An inconsistent actual procedure parameter

Procedure 'p' has a formal parameter 'q' which is specified as a procedure. It is possible to deduce from the use of 'q' inside 'p' that 'q' has one parameter of type <u>real</u> . When 'p' is called, it has an actual parameter 'r' which is a procedure with one <u>boolean</u> parameter. Thus the use of 'p' is inconsistent with its declaration and the program contains an error.

```
begin
        procedure r(b);
                boolean b;
                ;
        procedure p(q);
                procedure q;
                q(x);
        real x;
        p(r)
end
```

1.11 Declaration follows statement

```
begin
        procedure p(x);
                real x;
                ;
        ;
        real y;
end
```

1.12 A goto statement into a for statement

```
begin
        integer i;
        goto m;
        for i := 2 do
                begin
           m:
                end
end
```

1.13 Variables of different types in a left-part list

```
begin
        real x;
        integer i;
        x := i := 2
end
```

1.14 Invalid use of integer divide

```
begin
        integer i;
        i := 7;
        i := abs(i) div 2
end
```

1.15 A missing closing string quote

It is desirable that the programmer should be told not only that there is an unmatched string-quote-symbol at the end of his program, but also where the string starts.

```
begin
        procedure p ( s );
                string s;
                ;
        p ( '
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                0123456789abcdefghijklmnopqrstuvwxyz
                );
end
```

2. Programs which should fail during execution

2.1 Subscript outside the array bounds - (i)

```
begin
        real array a[ 1 : 10 ];
        a[0] := a[11]
end
```

2.2 Subscript outside array bounds - (ii)

```
begin
        array a[1 : 10, 1 : 3];
        a[2, 4] := 0
end
```

2.3 Subscript outside array bounds - (iii)

```
begin
        array a[1 : 3, 1 : 10];
        a[4, 2] := 0
end
```

2.4 Division by zero

```
begin
        real x;
        x := 3.7 / 0.0
end
```

2.5 Square root of a negative number

```
begin
        real x;
        x := sqrt( - 1.0)
end
```

2.6 Logarithm of zero

```
begin
        real x;
        x := ln(0.0)
end
```

2.7 Overflow on exponentiation

```
begin
        real x;
        x := exp(2000.0)
end
```

2.8 Use of a variable with no previous assignment

```
begin
        real x, y, z;
        x := y;
        x := z * z
end
```

2.9 The lower bound of an array exceeds the upper bound

```
begin
        array a[10 : 0];
        a[2] := 3
end
```

2.10 Zero exponentiate zero

```
begin
        integer i;
        i := 0;
        i := i ↑ i
end
```

2.11 -1.0 ↑ 2.0 is undefined

```
begin
        real x;
        x := - 1.0;
        x := x ↑ 2.0
end
```

2.12 Overflow during exponentiation

```
begin
        real x;
        x := 10 000;
        x := x ↑ 1 000
end
```

2.13 An infinite loop

This program is designed to test what happens to a progam which translates, but when executed goes into an infinite loop. It is undesirable for the programmer to get no information except 'fails, time limit'. It is equally undesirable that he should use his total allocation of computer time with this one program.

A retroactive trace often helps when trying to find the cause of an infinite loop because it can indicate the whole of the loop, and not just the single point in it where the program failed.

```
begin
  lab:
        goto lab
end
```

3. Programs which are legal but might contain an error

All these programs are valid ALGOL 60; each one should compile and run without failing. However, they all contain an odd feature which might be there only because the programmer has made an error. If the compiler produces a warning message about this odd feature, it may help the programmer to trace an otherwise troublesome mistake.

3.1 End comments

The programmer may have forgotten a semicolon after the first end . A helpful compiler will detect the error if it gives a warning of odd end comments.

```
begin
        integer x;
        if true then
                begin
                x := 1
                end
        x := exp(x)
end
```

3.2 Begin - end structure is invalid

The extra end in this program may indicate that an error has occurred earlier. A compiler will detect this error if it insists on a unique end-of-program symbol at the end of all ALGOL programs. It will also detect the error if it warns the programmer of extra text after the end of his program.

```
begin
        real x;
end
end
```

3.3 Assignment to a value parameter

The programmer may have been under a misapprehension when he made the assignment to the value parameter. The compiler should warn him that if he wants the result, the parameter must be called by name. If the compiler makes the assignment to the actual parameter (an error), then this program will fail subscript overflow.

```
begin
        procedure p(n);
                value n;  integer n;
                n := 100 000 * n;

        array a[0 : 1];
        integer n;
        n := 1;
        p(n);
        a[n] := 0
end
```

3.4 Real-to-integer operations are invisible

This program contains a real-to-integer operation. The compiler should warn the programmer:- (1) that he will lose accuracy; and (2) that to save time he should move the operation outside any inner loops.

```
begin
        integer i;
        i := 3.14
end
```

3.5 A null for loop

This program contains a for loop which is not executed; this is worthy of comment by the compiler (perhaps 'n' has an invalid value). The program is written so that if the loop is executed once (à la FORTRAN) the program will fail division overflow.

```
begin
        integer i, j, n;
        n := -3;
        for i := 0 step 1 until n do
                j := 2 div i
end
```

3.6 Identifier declared but not used

A program which contains an identifier which is declared but not used is probably longer than necessary. It may contain an error because the use of the identifier is misspelt.

```
begin
        real x, y;
        x := 1
end
```

3.7 Switch index overflow

This program does not contain an error according to the Revised Report. However in ECMA Subset ALGOL 60 (see ECMA, 1963) and IFIP Subset ALGOL 60 (see IFIP, 1964), a goto statement involving an undefined switch designator is undefined, i.e. an error. The program is sufficiently odd to warrant an error message from compilers dealing with strict ALGOL 60, and all other compilers should report a failure.

```
begin
        switch s := L1, L2, L1;
        goto s[4];
   L1: L2:
end
```

3.8 Real relations

Comparing two real values may well give different results with different compilers. It may help the programmer if he is warned whenever he does this.

```
begin
        real x, y;
        x := y := 2.0;
        if x = y then
                 ;
end
```

4. Programs to test the rigour of the compiler

4.1 Use of local identifier in array subscript bound

```
begin
        integer i;
        i := 3;
        begin
                array a[1 : i];
                a[1] := 0;
           i:
        end;
        i := 2
end
```

4.2 If, then, for, else ambiguity

This program is legal according to the ALGOL 60 Report, but not according to the Revised Report.

```
begin
        integer i, j;
        if true then
                for i := 2 do
                        j := 1
        else
                j := 1
end
```

4.3 Redeclaration of standard entity

This program is legal.

```
begin
        real sin;
end
```

4.4 A check that comments are correctly recognized

```
begin
        comment  an odd comment;
        real x;
        comment further than end, up to the semicolon in fact. real x;
        x := 1.0
end
```

4.5 Own arrays

This program checks both that <u>own</u> <u>array</u> is not a valid abbreviation for <u>own</u> <u>real</u> <u>array</u> , and whether dynamic own arrays are allowed.

```
begin
        integer i;
        i := 10;
        begin
                own array a[1 : i];
        end
end
```

4.6 Non printing characters are not significant

This program checks that space-symbols are not regarded as significant in the middle of identifiers or numbers.

```
begin
        boolean  acheckonlongidentifiers;
        real pi;
        a check on long identifiers := true;
        pi := 3.14159 26535 89793 23846 26433 83280;
end
```

THE ALGOL COMPILER TESTS

The compilers, machines and testers

The programs listed in the previous section have been executed on sixteen compilers in order to see how they treat incorrect programs. Some of the compilers tested were standard versions available as part of the manufacturer's software, others were produced in universities, etc. They are listed below together with the dates of the tests and the names of the people who performed them. Some compilers have since been improved, e.g. UNIVAC 1108.

ICL KDF9, Whetstone ALGOL Compiler, Miss R. Thorn, NPL, June 1970 (see Randell and Russell, 1964).

ICL 4120, The manufacturer's compiler, Miss R. Thorn, NPL, June 1970.

GE625, Honeywell Computer Time Sharing Service, Miss. R. Thorn, NPL, June 1970.

ALGOL W, Stanford University compiler on an IBM 360/67, E. Satterthwaite, Stanford University, Nov 1970 (see Wirth et al 1966, Bauer et al 1968 and 1971, Satterthwaite 1971).

ICL 1900, XABE (except programs 2.8, 3.1 which were run using XALT/3), R. L. Dees, ICL, June 1970.

UNIVAC 1108, The manufacturer's compiler, G. H. L. Buxton, NEL, June 1970.

IBM 360/65, The manufacturer's compiler, P. A. Samet, Joan Garrett, M. Thomas, University College, June 1970.

ICL ATLAS, The ALGOL compiler (6 Feb 1970), F. R. A. Hopgood, Atlas Computer Lab, July 1970.

XDS 9300, The manufacturer's compiler, I. D. Hill, Medical Research Council Computer Unit, July 1970.

IBM 7094/I, Alcor - Illinois 7090 compiler, E. Hansen, Atomic Energy Commission, Denmark, July 1970 (see Bayer et al 1967).

GIER4, GIER ALGOL III and ALGOL IV, E. Hansen, Atomic Energy Commission, Denmark, July 1970.

BABEL, The KDF9 Eldon Babel compiler, M. J. Parsons, NPL, Oct 1971 (see Scowen 1969).

EGDON, The KDF9 EGDON ALGOL compiler, B. Cooper and M. D. Poole, Culham Laboratory, July 1970.

ICL System 4/50, The manufacturer's compiler, C. Harris and M. D. Poole, Culham Laboratory, July 1971.

ALGOL 68R, RRE Compiler on ICL 1907F, I. Currie, Royal Radar Establishment, May 1971.

Electrologica X8 THE, THE ALGOL, C. Bron, Technological University, Eindhoven, October 1971.

The results

The results are summarized in four tables using the symbols:-

- T A translation error was detected
- R A runtime error was detected
- W A warning message was printed
- O No error was detected
- X The program was not tested with this compiler
- OK This legal program compiled and ran successfully
- NA There is no equivalent program in this version of ALGOL

No quantitative value for the debugging effectiveness of each compiler is given because these tests do not measure all the relevant factors. In any case the compilers were written for computers differing in age, size, cost, characteristics and operating systems.

I would be happy writing programs for any of the 7094, ALGOL 68R, ALGOL W or Babel compilers (the order is random). All these systems give clear error messages, check for all or most faults, and give a clear runtime post-mortem. All except 7094 give extra language features which simplify the expression of many programs. ALGOL W and Babel give a flowtrace and other tracing facilities. Babel and ALGOL 68R can be used from a terminal for online remote job entry and execution.

1. Translation errors

COMPILER	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	COMPILER
KDF9	T	T	T	R	T	R	R	T	R	R	T	T	T	T	T	KDF9
4100	T	T	T	T	T	T	T	T	T	T	T	T	T	T	(1)	4100
GE625	T	R	T	T	T	R	R	R	R	R	0	R	0	0	(1)	GE625
ALGOL W	T	T	T	T	T	T	(2)	T	T	R	T	T	T	(2)	T	ALGOL W
1900	T	T	T	T	T	T	R	T	0	0	T	0	X	X	X	1900
1108	T	T	T	T	R	R	R	R	R	(3)	0	0	(3)	0	0	1108
360/65	T	T	T	T	T	R	R	T	R	R	T	T	T	T	T	360/65
ATLAS	T	T	T	T	T	T	R	T	0	T	T	R	T	0	T	ATLAS
9300	T	T	T	T	R	(4)	(4)	R	R	(4)	T	(4)	0	T	T	9300
7094	T	T	T	T	T	T	T	T	T	0	T	T	T	0	T	7094
GIER4	T	T	T	T	T	T(5)	0	T	R(5)	0	T	0	T	T	(1)	GIER4
BABEL	T	T	T	T	T	T	T	T	T	(6)	T	T	T	(6)	T	BABEL
EGDON	T	T	T	T	T	T	0	T	0	T	T	T	T	T	T	EGDON
4/50	T	T	T	R(7)	T	T	0	T	0	T	T	T	T	T	T(7)	4/50
ALGOL68R	T	T	T	T	T	T	NA	T	NA	T	OK	T	OK	NA	T	ALGOL68R
X8 THE	T	T	T	T	R	R	R	R	R	R	T	T	T	0	T	X8 THE

(1) 4100, GE625, GIER4.
The Translator asks for more.

(2) ALGOL W.
The error in program 1.7 is found during translation if the formal parameter is specified to be REAL RESULT; otherwise the error is found during execution. Program 1.14 is a legal program in ALGOL W because 'abs' is an operator, but real operands for integer-divide fail during translation.

(3) 1108.
Program 1.10 fails at runtime because the declaration of 'x' must precede its use. Program 1.13 is legal in this version of ALGOL.

(4) 9300.
3.14 is assigned to the constant 2.71 in program 1.7. In program 1.6, <u>true</u> = 1 and <u>false</u> = 0. Program 1.10 probably fails to translate because 'x' is used before its declaration. Program 1.12 goes into a closed loop during execution.

(5) GIER4.
The errors in programs 1.6 and 1.9 are not detected in the GIER 3 compiler.

(6) BABEL.
There is no Babel program equivalent to 1.10. Program 1.14 is legal in Babel because 'abs' is an operator.

(7) 4/50.
Program 1.4 fails at runtime but gives no error message. The failure message for program 1.15 is a random alpha-numeric string.

2. Runtime errors

COMPILER	1	2	3	4	5	6	7	8	9	10	11	12	13	COMPILER
KDF9	R	R	0	R	R	R	R	R	R	R	R	R	(1)	KDF9
4100	R	R	R	R	R	(2)	R	0	R	0	R	(2)	(3)	4100
GE625	R	R	R	R	R	R	R	0	R	R	0	R	(3)	GE625
ALGOL W	R	R	R	R	R	R	R	0(4)	R(4)	0(4)	(9)	R	(1)	ALGOL W
1900	R	R	0	R	R	R	R	R(5)	R	X	X	X	X	1900
1108	R	R	R	0	R	R	R	(6)	R	R	0	R	(1)	1108
360/65	R	0	R	R	R	R	R	0	R	R	R	R	(1)	360/65
ATLAS	R	R	R	R	R	R	R	0	R	0	R	R	(1)	ATLAS
9300	R	R	R	R	R	R	R	0	T(7)	R	R	R	(7)	9300
7094	R	0	R	R	R	R	R	R	R	R	0	R	(1)	7094
GIER4	R	0	R	R	R	0	R	0	R	0	R	R	(3)	GIER4
BABEL	R	R	R	R	R	R	R	(6)	R	(8)	(9)	R	(1)	BABEL
EGDON	R	R	0	0	R	R	R	0	R	R	R	R	(1)	EGDON
4/50	0	0	0	R	R	R	R	0	R	R	R	R	(3)	4/50
ALGOL68R	R	R	R	R	R	R	R	R	R	X	(9)	R	(1)	ALGOL68R
X8 THE	R	R	R	0	0	0	0	0	R	0	0	0	(3)	X8 THE

(1) KDF9, ALGOL W, 1108, 360/65, ATLAS, 7094, BABEL, EGDON, ALGOL68R.
Program 2.13 fails time limit.

(2) 4100.
In program 2.6 the result is the largest possible negative number. Program 2.12 went into a runtime loop printing FPOFLO.

(3) 4100, GE625, GIER4, 4/50, X8 THE.
Program 2.13 continues until it is terminated by the operator.

(4) ALGOL W.
The postmortem dump indicates that no value has been assigned in program 2.8. The declaration in program 2.9 is legal but attempting to access an element fails. Program 2.10 is legal (per request of D. E. Knuth).

(5) 1900.
Program 2.8 fails overflow and the postmortem dump indicates that no value has been assigned.

(6) 1108, BABEL.
All variables are intialized to zero in these versions of ALGOL.

(7) 9300.
An error is found during translation in program 2.9 because the array has fixed bounds. Program 2.13 compiled, ran and terminated after about one minute without operator intervention.

(8) BABEL.
Program 2.10 fails during translation because the real exponentiate operator is defined only for 'real = real ↑

integer'.

(9) ALGOL W, BABEL, ALGOL68R.
Program 2.11 fails to translate in this version of ALGOL because real ↑ real is not defined in the language.

3. Warning messages

COMPILER	1	2	3	4	5	6	7	8	COMPILER
KDF9	W	T	0	0	0	W	R	0	KDF9
4100	0	0	0	0	0	0	R	0	4100
GE625	0	0	0	0	0	0	(1)	0	GE625
ALGOL W	T(2)	T	0	T	0	0	R	0	ALGOL W
1900	W	0	0	(3)	0	0	0	0	1900
1108	0	T	0	0	0	0	0	0	1108
360/65	W	T	0	0	0	0	R	0	360/65
ATLAS	(4)	0	0	0	0	0	0	0	ATLAS
9300	0	T	0	0	0	0	0	0	9300
7094	W	T	0	0	0	0	(5)	0	7094
GIER4	0	0	0	0	0	0	0	0	GIER4
BABEL	T(6)	T(6)	T(6)	T(6)	0	0	R(6)	0(6)	BABEL
EGDON	0	T	0	0	0	0	R	0	EGDON
4/50	T	0	0	0	0	0	R	0	4/50
ALGOL68R	NA	T	T	T	0	0	R	T	ALGOL68R
X8 THE	W	T	0	0	0	0	0	0	X8 THE

(1) GE625.
The compiler eventually reported 'system malfunction' for Program 3.7.

(2) ALGOL W.
The compiler detects the error in program 3.1 because only an identifier is allowed as an end comment.

(3) 1900.
The constant '3.14' is converted to '3' during translation in Program 3.4.

(4) ATLAS.
The compiler lists the text of program 3.1 with the end comment on the same line as end . An inadvertent error will thus be easier to spot.

(5) 7094.
If a switch overflow occurs (as in program 3.7), the program either stops or runs wild.

(6) BABEL.
Programs 3.1 to 3.4, 3.7 are illegal. End comments no longer exist; instead an end-of-line comment has been introduced, i.e. C <LIST OF BASIC SYMBOLS> NEWLINE-SYMBOL is equivalent to NEWLINE-SYMBOL. Each Babel program must finish with an end-of-program symbol. Assignment to value paramenters is not

permitted. Type conversion operators must be specified explicitly. The concept of switches has been replaced by Cases; a case-index overflow is defined as an error.

Each relation is preceded by a symbol which specifies the type of the operands, thus the real relation in program 3.8 is clearly indicated.

4. Other tests

COMPILER	1	2	3	4	5	6	COMPILER
KDF9	T	T	OK	OK	T	OK	KDF9
4100	0	0	T	OK	T	OK	4100
GE625	(1)	0	T	OK	T	OK	GE625
ALGOL W	(3)	T	OK	OK	(5)	T(2)	ALGOL W
1900	(3)	T	OK	X	X	X	1900
1108	T	0	OK	OK	0	T(12)	1108
360/65	W	T	OK	OK	(4)	W(6)	360/65
ATLAS	0	T	OK	OK	T	OK	ATLAS
9300	T	T	OK	OK	0	0(7)	9300
7094	0	0	OK	OK	0	OK	7094
GIER4	T	T(8)	OK	OK	(4)	T(8)	GIER4
BABEL	(3)	(9)	OK	OK	(5)	OK	BABEL
EGDON	T	0	OK	OK	T	OK	EGDON
4/50	T	T	OK	OK	T(10)	OK	4/50
ALGOL68R	(3)	NA	OK	NA(11)	(5)	X	ALGOL68R
X8 THE	T	T	OK	T	0	OK	X8 THE

(1) GE625.
Program 4.1 is inapplicable because array bounds must be constant in this version of ALGOL.

(2) ALGOL W.
Spaces are illegal within identifiers and constants.

(3) ALGOL W, 1900, BABEL, ALGOL 68R.
Program 4.1 is legal in these versions of ALGOL.

(4) 360/65, GIER4.
<u>own</u> is not implemented in these versions of ALGOL.

(5) ALGOL W, BABEL, ALGOL68R.
<u>own</u> is not a concept in these versions of ALGOL.

(6) 360/65.
For program 4.6, the compiler prints a warning message that the precision of the real constant beginning 3.1415926535 exceeds the internally handled precision and has been truncated.

(7) 9300.
The compiler makes no comment, but the value assigned to 'pi' is incorrect (= -6 * 10 ↑ -17).

(8) GIER4.

The error in program 4.2 is not detected in the GIER3 ALGOL compiler. In program 4.6 the constant is too long.

(9) BABEL.

Progam 4.2 is legal in Babel; the if -then statement has been replaced by a WHEN-DO statement. This change ensures that there are no ambiguities caused by a rule that a statement appearing in a syntax rule of Babel can always be any sort of statement.

(10) 4/50.

The compiler fails because own array is not the same as own real array . Dynamic own arrays are not implemented.

(11) ALGOL 68R.

Comments are always bracketed and may appear anywhere in ALGOL 68, so there is no need for a special end comment or parameter comment.

(12) 1108.

Spaces are illegal inside identifiers, and only 18 digits are allowed for a constant.

A SURVEY OF THE ERRORS MADE BY PROGRAMMERS

It does not seem very sensible to consider debugging facilities without knowing what errors are actually made by programmers. Accordingly, two of the KDF9 ALGOL 60 compilers used at NPL were modified (*1) to record details of the errors which were made. (*2)

There are two different ALGOL 60 compilers in use at NPL. One, known as WAlgol, is load-and-go with rapid compilation and interpretive execution; the other, known as KAlgol, compiles slowly but gives efficient code and is used for working programs. Two operating systems are used at NPL - Eldon provides interactive file editing and remote job entry facilities, and 'Red box' is the standard batch operating system used for all jobs which cannot be run under Eldon.

The programs surveyed at NPL are generally under development and cover a wide range of mathematical, scientific and engineering applications. Most users are familiar with ALGOL 60 because it has been the most commonly used language at NPL for the past seven years; however they are primarily scientists, not expert programmers.

Runtime errors

The Eldon WAlgol controller was modified so that it counted in a file the total number of programs which fail for each different error. This compiler is used when an ALGOL program is compiled and run from a teletype. There is a time limit of 30 seconds and the output must not exceed 4320 characters.

The results of surveying 8902 programs which failed during execution:-

No. of Failures	per cent	Reason for failure
4516	50.73	time limit
667	7.49	subscript overflow
544	6.11	call read at end of data
486	5.46	variable used before assignment
478	5.37	real overflow on /
397	4.46	array variable used before assignment
242	2.72	error in code body
223	2.51	actual - formal incompatibility
191	2.15	sqrt(x), x < 0
188	2.11	program needs too much space
182	2.04	read
147	1.65	real overflow not / or ↑
145	1.63	errors in stream number
109	1.22	error using ↑
95	1.07	dynamic type check

---

(1) I am grateful to A. L. Hillman and C. Knightley for making the alterations, and to D. Allin and M. J. Parsons for writing programs to summarize the results.
(2) L. B. Smith (see Smith, 1967) conducted a similar survey in 1966 to discover the errors that were made by students in six examples given on a programming course.

72	0.81	integer overflow not ↑
70	0.79	lower bound > upper bound
66	0.74	ln(x), x ≤ 0
35	0.39	write text
24	0.27	exp(x), x > 87
9	0.10	copy text
9	0.10	out of range switch
3	0.03	16 invalid basic symbols output
3	0.03	output a real number
1	0.01	read array

Notes on the runtime errors

The figures in the above table clearly depend on the compiler as much as the programs and programmers. Some errors are not detected (e.g. underflow); others are treated as merely worth a warning message (e.g. printing a number too big to fit into the specified format).

1. Time limit (51%)

This value is large because WAlgol is so slow that most programmers make no attempt to avoid it. They know that they must eventually run offline or use the KAlgol compiler to achieve faster execution.

2. Subscript overflow (7.5%)

This is a common failure. It is also a dangerous failure because an assignment to an illegal array element will overwrite some other variable, or, even worse, overwrite the code of the program. It is a time consuming check but there are various possibilities for reducing the inefficiency:-

(1) Do not check each subscript, but check only that the address of the subscripted variable lies somewhere in the array. Compilers which have detected an error in only one of the programs 2.2 and 2.3 presumably use this technique.

(2) Check array subscripts only on assignment; at least the completely disastrous effects of overwriting code are avoided (*1).

(3) Make the check a compiler option. This is dangerous because most programs thought to be correct still contain errors.

(4) The first three measures are palliatives; there are two better solutions. The first is to have special hardware to check subscripts built into the computer so that there is an interrupt whenever the check fails.

(5) A second possibility is to define and implement better programming languages which enable operations to be performed on complete and partial arrays. Subscript checks can then be replaced by less frequent checks that the arrays are

---

(1) A compiler for the ELX8 does this

compatible. APL, ALGOL 68 and PL/I are examples of languages which contain some of the required features. Note that such languages should also make it easier to avoid making the error.

3. Call read at the end of the data (6%)

This error is sometimes merely the result of poor programming. The frequency as an error may thus be exaggerated.

4. Overflow (9%)

Half of all the overflow errors occur because the programmer has divided by zero.

5. Using a variable before assignment (10%)

This common error is checked by very few compilers. Like subscript checking, it is expensive to check and most easily done by special hardware; failing this, it is probably best to make the check optional. Three new instructions would be necessary:-

(1) Store a special (= unassigned) value in a specified number of words.

(2) Cause an interrupt if any word with the unassigned value is accessed.

(3) Test if a given word has the unassigned value; this instruction is necessary in post mortem routines.

As with subscript overflow, there is an alternative solution of avoiding the error by redesigning the language, for example, every variable is assigned a value at its declaration. But there are disadvantages: block entry would be a very slow operation when large arrays are declared and it would still be rather difficult to trace the error when it occurs. Nevertheless, it is a good feature in FORTRAN and ALGOL 68 that it is possible to assign an initial value to a variable at its declaration. Good programming practice of using this option would reduce the frequency of the error.

6. Dynamic variable checks (3.5%)

Ideally it should be possible to check during translation that variables are always used in a manner consistent with their declaration; however it is difficult to make a complete check in ALGOL 60. Programs 1.4 to 1.10, 1.13, 1.14 test whether such errors are found during translation. Most ALGOL compilers, including WAlgol, are unsuccessful and need to make runtime checks. The successful compilers often compile only a subset of ALGOL 60.

The solution with this problem is definitely better language design; ALGOL W, ALGOL 68 and Babel compilers all check during translation that each use of an identifier is fully consistent with its declaration.

7. Errors while reading a number (2%)

WAlgol reads free-format data and is rather tolerant of incorrect data. A failure is detected if there is a syntax error in the data (e.g. two decimal points in a number or no digit after a decimal point) or if the number is too large. Other possible errors (e.g. too many digits, a silly value, reading out of step) must be checked by the programmer.

8. Output errors (.6%)

Output errors are rare because Walgol is very tolerant. When a value does not fit the specified format, it is printed with a default format. The layout of the results is spoiled but the results are those calculated: this is more sensible than preserving the layout by deleting leading digits. Similarly, when outputting symbols, a program fails only after the program has tried to output 16 non-existent symbols.

9. Errors in stream or channel numbers (1.3%)

These errors arise when the programmer:- (1) forgets to open a stream before using it, (2) closes it before the end of his program, (3) outputs to an input device or vice versa. A different form of Input Output scheme can be used which removes the possibility of making most of these errors. In this scheme all input and output goes to whichever suitable streams are currently specified in the program. Each program starts with one standard input and output device and for most programs this is sufficient for the whole program. A scheme like this has proved very convenient in Babel.

10. Errors in code bodies (3%)

It is not possible to deduce very much from the number of errors found in code bodies. WAlgol checks that the the stacks are not grossly incorrect at the end of a code procedure. A failure also occurs if the program executes an illegal instruction or jumps to an non-existent address. As with all machine code, it is easier to make mistakes than detect them.

11. Errors when calling a standard function (3%)

'sqrt' fails more frequently than 'ln' because it is called more often (see Wichmann, 1970).

12. Errors with dynamic array bounds (3%)

These errors occur when the upper bound of an array is so large that the array will not fit into the available space, or when an upper bound is less than the lower bound. The error 'program needs too much space' rarely occurs because a program is too complex (recursive) and has filled the stack.

These errors would not occur if the space required for program and arrays is found during translation (as in FORTRAN). I suspect the error often occurs in WAlgol because the bounds for the arrays have been read from incorrect data.

13. Switch index overflow (0.1%)

I am surprised how infrequently this error occurs; I assume that it is because most users do not understand and use switches. Perhaps when an error does occur, it is easily cured once and for all.

Translation errors

The 'red-box WAlgol' translator has been modified to record the translation failures. This is the batch compiler used for testing those programs which, for one reason or another, cannot be tested online in the Eldon system. For instance, they may be too big, or segmented, or use the graph plotter. Many of the errors are difficult to interpret and explain; they are probably caused by misprints.

For each program that fails to translate, the identifier and first four error messages are remembered. A program has been written to print the list of errors and to count the number of times each failure number occurs:- (1) as a first error, (2) as a subsequent error. Each entry in the table below specifies a translation failure, the number of programs in which it was the first failure reported, and the number of times it was reported as a second, third or fourth failure in a program.

The table gives only the most common errors. All other errors were reported as a first error in less than 18 programs.

Total number of programs that failed = 1383

FIRST	SUBSEQUENT	CAUSE OF ERROR
561	-	Identifier used but not declared or program is too large
92	84	Wrong number of subscripts or parameters
59	31	Redeclaration of identifier
54	31	No end-of-program symbol after program
45	34	End-of-program symbol inside program
43	37	Adjacent delimiters inadmissible
42	87	Current use of identifier is inconsistent with previous uses
41	20	Letter, digit, decimal point or subscript ten misplaced
32	7	Identifier in value-part or specification-part but not in formal-parameter-list, or vice versa
29	79	Illegal statement
24	12	Statement ends incorrectly
23	82	Declaration follows statement

Notes on the translation errors

The survey of translation errors has not been so successful because several

factors have added to the difficulty of analysing the figures:-

(1) Some of the errors are spurious, i.e. an earlier error has upset the compiler so that it reports one or more errors which do not actually exist. It is possible to recognize the errors which are probably spurious because they have a much higher frequency of being a subsequent error than a first error. For WAlgol I estimate that about a quarter of subsequent errors are spurious.

(2) The failure message usually says what the compiler found wrong, not the mistake made by the programmer. For instance, I suspect that the following errors are all very common:- deleting or inserting a character, transposing two characters, not underlining a basic symbol, confusion between I, l and 1, confusion between O (a letter) and 0 (a digit), typing a character in the wrong shift, missing out an operator, failing to match brackets. Although one of these errors usually causes a syntax error, the failure message does not give the error in this form.

(3) The two commonest errors with this compiler are 'identifier used but not declared' and 'program too large'; however, both errors have been remembered in the same way in the table.

1. Identifier used but not declared

This is by far the most common ALGOL 60 error. In WAlgol it is only detected if no other errors have been detected in the program.

In many programming languages (e.g. FORTRAN), variables do not have to be declared explicitly. In this case the error can be found easily only if the compiler prints a warning message.

2. Program is too large

The survey found this to be a common error only because the version of the compiler which was surveyed is mainly used for very large programs.

3. Inconsistent use of identifiers

WAlgol tries to check that identifiers are used consistently with their declaration; however, the checks are not complete and as a result some errors are not detected until execution.

4. Begin - end structure

Errors in the begin - end structure of a program show up in a variety of ways: an error in a procedure body will appear as 'declaration follows statement', an extra begin (or opening string quote) as end-of-program symbol inside program, an extra end as 'no end-of-program symbol after program'.

5. The number of translation errors found at a time

WAlgol is quite successful at finding all the syntax errors in a program; therefore the number of errors reported will be approximately the same as the number of syntax errors in the program.

Programs	Number and sort of errors
40%	One or more identifiers used but not declared or program is too large
30%	One syntax error
12%	Two syntax errors
5%	Three syntax errors
13%	Four or more syntax errors

CONCLUSION

Good debugging facilities do not arise by chance, but through the foresight of the software engineer when he was designing the compiler.

This paper has outlined some compiler properties and available tools which aid debugging, and shown that some compilers are vastly superior to others. Debugging is important because incorrect programs are expensive.

Not enough attention has been paid to the problems of ensuring that a program does what was originally intended. As a result some languages are far better than others at guiding the programmer to produce correct programs. A good language for debugging contains redundancy so that a random misprint almost always results in a syntactically incorrect program; a good language also has a natural compact notation.

Compilers also differ greatly, even for the same language. A compiler makes debugging easy if it is small and fast, produces code which is at least reasonably efficient, has a very rapid turnround, and gives concise clear error messages. A compiler should also provide extra documentation and diagnostic aids.

ACKNOWLEDGEMENTS

I am grateful to colleagues at NPL who helped this survey in various ways including discussions, amending compilers, writing and running various programs, typing results, etc; among them are D. Allin, Miss L. Ellis, A. L. Hillman, C. Knightley, M. J. Parsons, Miss. H. Pinkham, Miss R. Thorn and Dr. B. A. Wichmann.

I am also grateful to all those people (see page 19) who took the trouble to run the test programs and let me have the results.

Finally I would also like to thank I. D. Hill for suggesting some of the test programs.

REFERENCES

Anon,
Figure 17 - Computer output can have many variations and can keep students occupied for hours,
SIGPLAN Notices 4, 11(Nov 1969), 20.

J. M. Adams, J. B. Johnston, R. H. Start, (Editors),
Proceedings of an ACM Conference on 'Proving assertions about programs',
SIGPLAN Notices, 7, 1(Jan 1972).

H. Bauer, S. Becker, S. Graham,
ALGOL W Implementation,
Stanford University Computer Science Report, CS 98, May 1968.

H. Bauer, S. Becker, S.L. Graham, E. Satterthwaite, R.L. Sites,
ALGOL W Language definition,
Stanford University Computer Science Report CS 230, July 1971.

R. Bayer, D. Gries, M. Paul, H. R. Wiehle,
The ALGOL Illinois 7090/7094 Post Mortem Dump,
Comm ACM, 10(Dec 1967), 804 - 808.

P. J. Brown,
ML/I User's manual,
University Mathematical Laboratory, Cambridge, England, July 1966.

P. J. Brown,
The ML/I Macro processor, Comm ACM, 10(Oct 1967), 618 - 623.

K. Conrow, R. G. Smith,
NEATER 2, A PL/I Source Statement Reformatter,
Comm ACM, 13(Nov 1970), 669 - 674.

ECMA,
ECMA subset of ALGOL 60,
Comm ACM, 6, 10(Oct 1963), 595-597.

B. Elspas, M. W. Green, K. N. Levitt,
Software reliability, Computer (IEEE), Jan-Feb 1971, 21-27.

D. G. Evershed, G. E. Rippon,
Highlevel languages for lowlevel users, Comp. J, 14, 1(Feb 1971), 87 - 90.

G. E. Forsythe,
Pitfalls in computation, or why a math book isn't enough,
Stanford University Computer Science Report, CS 147, Jan 1970.

F. Gruenberger,
Problems and priorities,

Datamation, Vol 18, 3(Mar 1972), 47 - 50.

IFIP,
Report on SUBSET ALGOL 60(IFIP),
Comm ACM, 7, 10(Oct 1964), 626 - 628.

R.L. London,
Proving programs correct - some techniques and examples,
BIT, 10, 2(1970) 168 - 182.

P. G. Moulton, M. F. Muller,
DITRAN - A compiler emphasizing diagnostics, Comm ACM, 10, 1(Jan 1967), 45 - 52.

B. Randell and L. J. Russell,
'ALGOL 60 Implementation',
Academic Press, 1964.

E. Satterthwaite,
Debugging tools for high-level languages,
Computing Laboratory Technical Report 29, Newcastle-upon-Tyne University, Dec 1971.

R. S. Scowen,
Babel, a new programming language, NPL CCU Report No 7, Oct 1969.

R. S. Scowen, D. Allin, A. L. Hillman, M. Shimell,
SOAP - A program which documents and edits ALGOL 60 programs,
Comp J, pp133 - 135, Vol 14, No 2, 1971.

P.W. Shantz, R.A. German, J.G. Mitchell, R.S.K. Shirley, C.R. Zarnke,
WATFOR - The University of Waterloo FORTRAN IV compiler,
Comm ACM, 10, 1(Jan 1967), 41 - 44.

L. B. Smith,
Part one: A comparison of batch processing and instant turnround. Part two: A survey of most frequent syntax and execution time errors, Stanford Computation Center, Feb 1967.

E. C. Van Horn,
Three criteria for designing computing systems to facilitate debugging,
Comm ACM 11, 5(May 1968), 360 - 365.

N. Wirth, C. A. R. Hoare,
A contribution to the development of ALGOL, Comm ACM, pp413 - 432, Vol 9, No 6, 1966.